<u>CS147 – Project 02 Implementation Guide line.</u>

1. The project is to implement a behavioral model, using verilog, of a processor supporting CS147DV.
2. Starter code is supplied contains skeleton implementation of a computer system and corresponding testbench to test the system. The files are as the following.
   - 2.1. da_vinci.v : Complete system integrating processor and 32x64M memory.
   - 2.2. memory.v : Parameterized memory model of a 32x64M memory.
   - 2.3. processor.v : Processor implementation integrating CU, RF and ALU.
   - 2.4. alu.v : Model of ALU using in this processor.
   - 2.5. register_file.v : 32x32 register file (RF) model.
   - 2.6. control_unit.v : The control unit (CU) of the process.
   - 2.7. mem_64MB_tb.v : Testbench source to test the memory model.
   - 2.8. da_vinci_tb.v : Testbench source to test the DaVinci 1.0 system.
3. Most of the source codes are already completed (to save you from tedious job of connecting ports and wires in the source file). The memory model and test have been implemented completely in memory.v file for your reference for register file implementation and testing. You need to modify the core functionality of the the following modules.
   - 3.1. ALU : alu.v
     - a) Add registers for corresponding output ports.
     - b) Copy functionality from project 1. Need to change the port names to match the latest port naming.
     - c) Extend the functionality to set the 'ZERO' output, which is set to 1 if result of the current ALU operation is zero. It is set to 1 otherwise.
     - d) Add a testbench file to test ALU.s
   - 3.2. RF : register_file.v (very much alike memory.v)
     - a) Add registers for corresponding output ports.
     - b) Add 32x32 memory storage similar to 32x64M memory in memory.v
     - c) Add 'initial' block for initializing content of all 32 registers as 0.
     - d) Register block is reset on a negative edge of RST signal.
     - e) On read request, both the content from address ADDR_R1 and ADDR_R2 I returned.
     - f) On write request, ADDR_W content is modified to DATA_W.

g) Do not handle read,write = 00 or 11. In that way, for such configuration, the RF hold the previously read data.

h) Write a testbench similar to memory (do not need to load the RF externally though like the memory testing). Just write some data in all the register location, and read them back.

3.3. CU: control_unit.v

a) Add registers for corresponding output ports.

b) State machine implementation (refer to lab08 source code for writing state machine)

- Modify module 'module PROC_SM(STATE,CLK,RST);'
- Define state and next_state register.
- State machine is reset on negative edge of RST. At reset and 'initial' set the next state as `PROC_FETCH (defined in proj_definition.v) and state to 2 bit unknown (2'bxx).
- Upon each positive edge of clock, assign state with next_state value. Also determine next_state depending on current state value.
  - **`PROC_FETCH  → `PROC_DECODE;**
  - **`PROC_DECODE → `PROC_EXE;**
  - **`PROC_EXE  → `PROC_MEM;**
  - **`PROC_MEM  → `PROC_WB;**
  - **`PROC_WB  → `PROC_FETCH;**

c) Control signal generation.

- Add registers for corresponding output ports.
- Define a register for write data to memory and connect DATA to this register similar to the DATA port in memory model (but in opposite sense – in control unit, for memory read operation DATA must be set to HighZ and for write operation DATA must be set to internal write data register).
- Add internal registers as following
  - **PC_REG** : Holds program counter value
  - **INST_REG** : Stores the current instruction
  - **SP_REF** : Stack pointer register
- Always at the change of processor state set the control signal and address / data

signal values properly for memory, register file and ALU.

- **`PROC_FETCH** : Set memory address to program counter, memory control for read operation. Also set the register file control to hold ({r,w} to 00 or 11) operation.
- **`PROC_DECODE** : Store the memory read data into INST_REG; You may use the following task code to print the current instruction fetched (usage: print_instruction(INST_REG);). It also shows how the instruction is parsed into required fields (opcode, rs, rt, rd, shamt, funct, immediate, address for different type of instructions). You may want to calculate and store sign extended value of immediate, zero extended value of immediate, LUI value for I-type instruction. For J-type instruction it would be good to store the 32-bit jump address from the address field. Also set the read address of RF as rs and rt field value with RF operation set to reading.

```
task print_instruction;
input [`DATA_INDEX_LIMIT:0] inst;

reg [5:0]   opcode;
reg [4:0]   rs;
reg [4:0]   rt;
reg [4:0]   rd;
reg [4:0]   shamt;
reg [5:0]   funct;
reg [15:0]  immediate;
reg [25:0]  address;

begin
// parse the instruction
// R-type
{opcode, rs, rt, rd, shamt, funct} = inst;
// I-type
{opcode, rs, rt, immediate } = inst;
// J-type
{opcode, address} = inst;

$write("@ %6dns -> [0X%08h] ", $time, inst);

case(opcode)
// R-Type
6'h00 : begin
            case(funct)
                6'h20: $write("add  r[%02d], r[%02d], r[%02d];", rs, rt, rd);
                6'h22: $write("sub  r[%02d], r[%02d], r[%02d];", rs, rt, rd);
                6'h2c: $write("mul  r[%02d], r[%02d], r[%02d];", rs, rt, rd);
                6'h24: $write("and  r[%02d], r[%02d], r[%02d];", rs, rt, rd);
                6'h25: $write("or   r[%02d], r[%02d], r[%02d];", rs, rt, rd);
                6'h27: $write("nor  r[%02d], r[%02d], r[%02d];", rs, rt, rd);
                6'h2a: $write("slt  r[%02d], r[%02d], r[%02d];", rs, rt, rd);
                6'h00: $write("sll  r[%02d], %2d, r[%02d];", rs, shamt, rd);
                6'h02: $write("srl  r[%02d], 0X%02h, r[%02d];", rs, shamt, rd);
                6'h08: $write("jr   r[%02d];", rs);
                default: $write("");
            endcase
        end
// I-type
6'h08 : $write("addi  r[%02d], r[%02d], 0X%04h;", rs, rt, immediate);
6'h1d : $write("muli  r[%02d], r[%02d], 0X%04h;", rs, rt, immediate);
6'h0c : $write("andi  r[%02d], r[%02d], 0X%04h;", rs, rt, immediate);
6'h0d : $write("ori   r[%02d], r[%02d], 0X%04h;", rs, rt, immediate);
6'h0f : $write("lui   r[%02d], 0X%04h;", rt, immediate);
6'h0a : $write("slti  r[%02d], r[%02d], 0X%04h;", rs, rt, immediate);
6'h04 : $write("beq   r[%02d], r[%02d], 0X%04h;", rs, rt, immediate);
6'h05 : $write("bne   r[%02d], r[%02d], 0X%04h;", rs, rt, immediate);
6'h23 : $write("lw    r[%02d], r[%02d], 0X%04h;", rs, rt, immediate);
6'h2b : $write("sw    r[%02d], r[%02d], 0X%04h;", rs, rt, immediate);
// J-Type
6'h02 : $write("jmp   0X%07h;", address);
6'h03 : $write("jal   0X%07h;", address);
6'h1b : $write("push;");
6'h1c : $write("pop;");
default: $write("");
endcase
$write("\n");
end
endtask
```

- **`PROC_EXE** : In this stage, set the ALU operands and operation code accordingly the opcode/funct of the instruction. Some operation may not need ALU operation (like lui, jmp or jal). In those cases, just do not put anything on the operands or operation. For 'push' operation, the RF ADDR_R1 needs to be set to 0 (because push stores the data at r[0] into the stack).
- **`PROC_MEM**: Only 'lw', 'sw', 'push' and 'pop' instructions are involved in this stage. By default, make the memory operation to 00 or 11 (which will set the memory output to HighZ). For the four memory related operation, set the memory read or write mode accordingly (e.g. for read you need to set MEM_READ to 1'b1 and MEM_WRITE to 1'b0). The Address for the stack operation needs to be set carefully following the ISA specification.
- **`PROC_WB**: Write back to RF or PC _REG is done here.
  - Increase PC_REG by 1 by default.

    - Reset the memory write signal to no-op (00 or 11)
    - Set RF writing address, data and control to write back into register file. For most of the instruction this is needed.
    - The instructions beq, bne, jmp and jal needs to modify the PC_REG accordingly the instruction specification.

- To test whole system, create memory data file with instruction memory content similar to 'fibonacci.dat' file and pass this file into the system by modifying the data file name at line 38 of da_vinci_tb.v. Please check (and change if needed) the memory dump file name and location at line 51 of the same file (using $writememh system task) to capture relevant memory location (where your test program is writing its result). You may also dump the register file similarly if needed.